Enhanced Sampling Toolkit Documentation

Release 0.1a

Jeremy O. B. Tempkin

Jun 07, 2016

CONTENTS

CHAPTER

ONE

INTRODUCTION

The objective of the Enhanced Sampling Toolkit is to provide a programming toolkit that facilitates rapid prototyping and development of enhanced sampling algorithms for use in molecular dynamics applications.

The toolkit is implemented in 100% Python and is targeted for development in the Python language. One reason for this decision is that Python is a popular language in the broader scientific community and has a wide support base for developing scientific codes. However, more importantly, a strength of the language lies in the speed at which ideas can be implemented into working code. Since rapid and flexible prototyping of new algorithms is the core priority of the toolkit, Python seems a natural choice in language. However, if the need arises in a future date, ports to other languages may be considered and integrated into the package.

In many places of the toolkit some care has been made to adhere to the powerful Python object design principles inherent in the Python data model. We find this to be a powerful feature of the Python language since it facilitates clean, "Pythonic" use of the toolkit in our applications. We are continually working to improve this aspect of the toolkit as the project develops.

In the basic sense, the toolkit serves to wrap commonly used MD codes and in the process abstract the interactions between algorithm level code and the MD engine. This abstractions provides useful extensibilty in the sense that algorithm codes that are implemented in the Walker API can be reused and swapped between MD models and even entire MD codes. Furthermore, the expensive integration steps are executed in faster compiled codes and avoids some of the inefficiencies introduced in the choice of Python.

In addition to rapid algorithm prototyping, we've found in it's development that the toolkit is effective in HPC environments as well. Because the underlying dynamics are executed in commonly used MD codes, the toolkit has access to the HPC features that have been optimized in these codes and can therefore leverage MPI parallelism concurrently at the algorithm level and the MD level as well as support for accelerators such as GPU's or Intel MIC cards.

The structure of the toolkit comprises of two parts. At the core, we provide a specification we call the Walker API. The core API serves to define a set of interactions between algorithm level code and the underlying MD engine. This specification also defines the features developers need to implement for a new dynamics engine to leverage the full features of the toolkit.

On top of this core API, we've implemented several common enhanced sampling algorithms.

CHAPTER

TWO

WALKER API

The central design principle of the Walker API is to provide an abstracted interface between the "algorithm" level code and the code that integrates the dynamics of the model. This section describes the base API used to define the walker objects. The base class is declared internally using Python's abstract base class module. The implementations of the bindings specific to the dynamics packages are registered to this base class definition through direct subclassing of the modules. A key feature of this decision is that it enforces the implementation of the dynamics bindings to implement each of the following methods in their class declarations. However, many of the methods can be overridden with empty methods if an implementation is incomplete or does not require all of the base class methods.

2.1 The Core Walker Application Programming Interface (API)

DEVELOPER NOTE: we may want to use abstract properties in the future.

class walker_base.**walker**

This object acts as an abstract base class that sets the core specification defining the Walker API. These routines

addColvars(*cv*)

Adds a new collective variable to the walker. Subsequent calls to routines that act on the defined collective variables will use the complete list of collective variables in the order they are added. To wipe this list of variables, see the destroyColvars() routine.

close()

Destroys the walker.

destroyColvars()

Removes all collective variables from the walker.

drawVel()

Draws a new value of the velocities for the walker. Can redraw velocities according to a uniform or Gaussian distributions.

equilibrate(*center*, *restraint*, *numSteps*)

getColvars()

This function returns the location of the walker in the collective variable space. Values are returned as a one-dimensional numpy array

getConfig()

This function should return the postition of the walker in configuration space as a one-dimensional numpy array. The coordinates are returned as [x1, y1, z1, x2, y2, z2, ...].

getVel()

This function returns the velocities of the system as a one-dimensional numpy array. The velocities are

returned in a format that matches the getConfig() routine, specifically [v_x1, v_y1, v_z1, v_x2, v_y2, v_z2, ...].

propagate(*numsteps*)
Integrates the dynamics of the model forward in time. Takes the numSteps argument specifying the number of time steps to take.

removeOutput()
This routine adds a source of output for the walker to write information to disk.

reverseVel()
This function reverses the velocity of the walker. The new velocities are given as

setConfig(*configuration*)
Sets the configuration of the At minimum, it should take some sort of specification of the configuration.

setOutput()
This routine adds a source of output for the walker to write information to disk.

setVel()
This routine sets the velocities of the system.

2.1.1 Dynamics Modules

This module contains definitions of the dynamics classes using to wrap information about driving dynamics inside the walkers.

class dynamics.**baoab**(*temperature*, *damping_coefficient*, *shake=False*, *seed=None*, *linear_momentum=True*)
This class implements the baoab integrator.

class dynamics.**langevin**(*temperature*, *damping_coefficient*, *shake=False*, *seed=None*, *linear_momentum=True*)
This class specifies the langevin dynamcis parameters needed.

2.1.2 Setting Walker Output Files

Created on Thu Apr 16 15:03:56 2015

@author: jeremytempkin

class outputClass.**outputClass**(*name*, *outputType*, *filename*, *nSteps=1000*)
This class acts a container object for an output for a walker object.

2.1.3 Setting Walker Collective Variables

A container class for collective variable definitions. We provide here a container class for

2.2 LAMMPS Walker Module

This module implements the Walker API for the LAMMPS MD engine. See walker_base.py for a specification of the API. For details concerning the usage of the LAMMPS MD package see the excellent documentation at the LAMMPS webpage:

http://lammps.sandia.gov/

In particular, you may want to see how the Python wrapper to LAMMPS on which this implementation is based:

http://lammps.sandia.gov/doc/Section_python.html

Here we will outline basic usage guides for the walker API usage in LAMMPS. This module implements the Walker API for the LAMMPS MD engine. See walker_base.py for a specification of the API. For details concerning the usage of the LAMMPS MD package see the excellent documentation at the LAMMPS webpage:

http://lammps.sandia.gov/

In particular, you may want to see how the Python wrapper to LAMMPS on which this implementation is based:

http://lammps.sandia.gov/doc/Section_python.html

Here we will outline basic usage guides for the walker API usage in LAMMPS. To begin using the lammpsWalker module

- highlight here an example code which shows how to use the LAMMPS walker module.

class `lammpsWalker.`**`lammpsWalker`**(*inputFilename*, *logFilename*, *index=0*, *debug=False*)

This class implements the enhanced sampling walker API for the bindings to the LAMMPS package.

Some usage issues to note:

1) Collective variables (CVs) are defined to the walker by constructing a list of CVs internally in the walker.colvars object. These CVs list takes the following format:

["coordinateType", atomids....]

The coordinate type specifies which type of coordinate the CV is (i.e. bond, angle, dihedral, etc.). The next items in the list are the atom indecies involved in this specific instance of the CV.

The walker will use this list to initialize them to the underlying LAMMPS objects.

`addColvars`(*name*, *cvType*, *atomIDs*)

Implements the addition of a collective variable to the list of collective variables held by this walker.

name - A internal string used to reference this collective variable. Collective variable names must be unique.

cvType -

A string refering to the type of collective variable. Currently the following variables are supported:

- 'bond'
- 'angle'
- 'dihedral'
- 'x', 'y', 'z' positions
- 'x', 'y', 'z' velocity components

atomIDs -

A list of the atom indecies involved in the collective variable. Should provide the right number of atom indecies f

- 'bond' -> 2
- 'angle'-> 3
- 'dihedral' -> 4
- position or velocity component -> 1

close()
This function closes the LAMMPS object.

command(*command*)
This function allows the user to issue a LAMMPS command directly to the LAMMPS object.

destroyColvars()
This function removes the colvars set by setColvars(). By default, it removes all of the collective variables in the list. It does not remove them from the collective variables list.

drawVel(*distType='gaussian'*, *temperature=310.0*, *seed=None*)
This function redraws the velocities from a maxwell-boltzmann dist.

equilibrate(*center*, *restraint*, *numSteps*)
This function prepares a LAMMPS image to be at the specified target position given by the vector 'center' passed and an arguments.

getColvars()
This function returns the current position of the LAMMPS simulation in colvars space.

getConfig()
This function returns the current position of the LAMMPS simulation.

getVel()
This function returns the current velocities from the LAMMPS simulation.

get_time()
Return the time associated with the state of the walker.

minimize(*args=None*)
This function runs a minimization routine with the specified type.

propagate(*numSteps*, *pre='no'*, *post='no'*)
This function issues a run command to the underlying dynamics to propagate the dynamics a given number of steps.

removeOutput()
This routine removes the ouput pipes for information to be written to disk from the underlying dynamics engine. Right now this simply clears all existing output.

reverseVel()
This function reverses the velocities of a given LAMMPS simulation

setConfig(*config*)
This routine sets the internal configuration.

setDynamics(*dynamics_instance*)
This routine sets the dynamics for the walker.

setOutput(*name*, *outputType*, *filename*, *nSteps*)
This routine sets up a mechanism for writing system information to file directly from the dynamics engine. This is equivalent to output constructed

setTemperature(*temp*)
This function sets the temperature of the walker object.

setTimestep(*timestep*)
This routine sets the dynamics time step.

setVel(*vel*)
This function sets the velocity to the lammps simulation.

set_time(*t*)
Set the time of the walker.

2.3 OpenMM Walker Module

We plan to implement an OpenMM walker API module in a future release.

CHAPTER

THREE

APPLICATIONS

3.1 Nonequilibrium Umbrella Sampling

The Nonequilibrium Umbrella Sampling (NEUS) application module provides a flexible set of tools built on top of the Walker API that implements the found in **Tempkin et at, in prep**.

Put here a basic overview of the types of calculations one can do using this code base with a reference to the testable code in the examples/ subdirectory.

- A summary of some of the results or a summary of a basic implementation could be really useful here.
- Include a LAMMPS walker implementation of NEUS on the di

3.1.1 Partition Module Basic Usage

The partition module provides a simple structure for handling and manipulating a list of windows. The partition module is effectively a callable Python list.

To initialize a partition object:

```
import partition
import numpy as np
sys = partition.partition()
```

One can add elements to the partition one by one:

```
# import a window object with a pyramid support
import pyramid
win = pyramid.Pyramid([0.0], [1.0])

sys.append(win)

print sys
```

```
partition([Pyramid([ 0.], [ 1.]), Pyramid([ 0.], [ 1.]), Pyramid([ 0.], [ 1.])])
```

Or hand partition a list at initialization:

```
windows = [pyramid.Pyramid(i,[1.0]) for i in np.linspace(0.0, 10.0, 11)]
sys = partition.partition(windows)
print sys
```

```
partition([Pyramid(0.0, [ 1.]), Pyramid(1.0, [ 1.]), Pyramid(2.0, [ 1.]), Pyramid(3.0, [ 1.]), Pyram
```

Further, partition supports the addition operation for merging two partition objects:

```
#union = sys1 + sys2
#print union
```

Partition Class

class partition.**partition**(*umbrellas=[]*)

__call__(*pos*)
Return a numpy array of the normalized supports of the windows at the given.

__init__(*umbrellas=[]*)
Construct an instance of the partition.

append(*value*)
Append a new value to the partition.

Note that this enforces that the input object is callable. Will throw an exception if value is not callable.

Parameters **value** (*obj*) – A callable Python object.

Returns

Return type None

index(*item*)
Return the first index in which item appears in partition.

Extended description.

Parameters **item** (*obj*) – The window object to check for membership.

Returns **index** – The index of item in the list. Returns ValueError if item is not contained in the partition.

Return type int

3.1.2 Window Module

The window object defines a single spatiotemporal restriction of the discretization defined by the NEUS J(t) process. An instance of the window object corresponds to a single value of the J. The object defines a data structure for storing the entry point flux lists defined by ~pi. It also defintes the routines require for constructing the flux lists and returning physically weighted elements.

To import and initialize a window object:

```
>>> from est.neus import Window
>>> win = Window.window(center=[], width=[])
>>> print "Hello"
```

To An implementation of the window object in the NEUS toolkit.

class window.**window**(*center*, *width*, *ref_center=None*, *ref_width=None*, *time=None*, *periodic_length=None*, *max_list_size=100*, *initial_conditions=[]*, *initial_conditions_probability=0.0*, *a=0.0*)
The window object defines a single piece of the discretization defined by the NEUS J(t) process. An instance of the window object corresponds to a single value of the J. The object defines a data structure for storing the entry point flux lists defined by ~pi. It also defines a routine for returning an entry point based on the defined fluxes.

This object contains the following items:

- data structure for containing flux lists
- data structure for the initial conditions distribution at J(0)
- a list of physical fluxes used to draw entry points proportionally
- routines for reinjection
- routines for updating fluxes
- routines for updating entry point lists

add_entry_point(*ep*, *key*)
Add a new entry point to the specified flux list. If key is not already know, create a new set with label "key" and add it to the dictionary.

clear_flux_list()
Clear the flux list of all entries.

get_entry_point()
Return an entry point from the store flux lists proportional to the entry point fluxes.

get_flux_lists()
Return the dictionary of the flux lists.

get_initial_conditions()
Return the distribution of initial conditions.

reinject()
Return an entry point drawn proportional to the neighbor fluxes.

set_initial_conditions(*distribution*)
Set the distribution of initial conditions to this window. This corresponds the distribution of the process at J(0). Distribution is expected to be an iterable of entry point objects.

set_initial_conditions_probability(*p*)
Set the probability for drawing from the initial conditions.

update_fluxes(*fluxes*)
Set neighbor fluxes as numpy array.

The Pyramid module implements a support function for the Window class of the form:

equation.

Pyramid Class

Class definition for Pyramid shaped window. Inherets from Window.

class pyramid.**Pyramid**(*center*, *width*, *ref_center=None*, *ref_width=None*, *time=None*, *periodic_length=None*, *max_list_size=100*, *initial_conditions=[]*, *initial_conditions_probability=0.0*, *a=0.0*)
Class definition of Pyramid which inherits from window. Implements the pyramid shaped basis function.

ref_indicator(*coord*)
Return the value of the indicator for the reference.

time_indicator(*coord*)
Return the value of the indicator function for the time coordinate.

The Box module implements a support function for the Window class of the form:

Box indicator function.

3.1.3 Entry Point Module

This module acts as a template for defining named tuples that store entry points.

It provides a consistent entry point contruction as a named tuple.

It's provided as an independent file so that analysis and manipulation of these objects can be possible.

class `entryPoints.`**`entry_point`**(*q*, *p*, *ref_q*, *ref_p*, *time*, *cv*)

`cv`
Alias for field number 5

`p`
Alias for field number 1

`q`
Alias for field number 0

`ref_p`
Alias for field number 3

`ref_q`
Alias for field number 2

`time`
Alias for field number 4

CHAPTER

FOUR

TESTING

Here I will add a description of the unit testing of Walker API implementations and potentially the application testing.

To add: automatic system for scanning the package files for tests to execute. The idea here would be that if contributors to the project add test files the testing suite will automatically find and run these test files.

c

d

e

l

o

p

w

Symbols

A

B

C

D

E

G

I

L

M

O

P

Q

R

S

T

U

W